Parallel Local Search on GPU

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallel Local Search on GPU

Thé Van Luong — Nouredine Melab
— El-Ghazali Talbi

N° 6915

Mai 2009

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6915--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche LILLE - NORD EUROPE

Parallel Local Search on GPU

Th Van Luong, Nouredine Melab
, El-Ghazali Talbi

Thème NUM — Systèmes numériques
Équipe-Projet Dolphin

Rapport de recherche n° 6915 — Mai 2009 — 24 pages

Abstract: Local search algorithms are a class of algorithms to solve complex optimization problems in science and industry. Even if these efficient iterative methods allow to significantly reduce the computational time of the solution exploration space, the iterative process remains costly when very large problem instances are dealt with. As a solution, graphics processing units (GPUs) represent an efficient alternative for calculations instead of traditional CPU. This paper presents a new methodology to design and implement local search algorithms on GPU. Methods such as tabu search, hill climbing or iterated local search present similar concepts that can be parallelized on GPU and then a general cooperative model can be highlighted. In addition, this model can be extended with a hybrid multi-core and multi-GPU approach for multiple local search methods such as multistart. The conclusions from both GPU and multi-GPU experiments indicate significant speed-ups compared to CPU approaches.

Key-words: graphics processing units, local search, metaheuristics, parallel computing

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Parallel Local Search on GPU

Résumé : Les algorithmes de recherche locale sont une classe d'algorithmes pour résoudre des problèmes d'optimisation complexes en sciences et dans l'industrie. Même si ces méthodes iteratives efficaces permettent de réduire de manière significative le temps de calcul de l'exploration de l'espace de recherche d'une solution, ce dernier reste coûteux lorsque de très grandes instances d'un problème sont traitées. Pour résoudre cela, l'utilisation des cartes graphiques semble être une intéressante alternative aux calculs utilisant les processeurs habituels. Ce papier présente une nouvelle méthodologie pour concevoir et implémenter les métaheuristiques à solution unique sur carte graphique tels que la recherche tabou, la descente du gradient ou encore la recherche locale itérée. D'un point de vue conceptuel, cette classe d'algorithmes présentent beaucoup de points communs qui peuvent être parallélisés. On peut ainsi mettre en évidence un certain modèle général coopératif. Ce modle peut tre aussi tendu avec une approche hybride multi-coeur multi-GPU pour les algorithmes de recherches locales multiple. Les conclusions des expériences montrent des accélérations significatives lorsque la carte graphique est utilisée comme coprocesseur.

Mots-clés : processeur graphique, recherche locale, métaheuristiques, calcul parallèle

Contents

1 Introduction

GPU is a dedicated graphics rendering device for manipulating and displaying computer graphics. Recent GPU cards provide highly parallel structure with high execution capabilities and fast memory access. Since the same program is executed on many data elements in GPU and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Some computational tasks that previously would have taken long computations become interactive, because computational time can be significantly reduced with recent graphic cards. As a consequence, the use of GPU for an efficient implementation of metaheuristics is becoming a challenging issue. Indeed, the main difficulty on GPU reside in managing memory transactions. In contrast to CPU, GPUs require massively data-parallel computations with predictable memory accesses.

Previous works on population based metaheuristics on GPU has been proposed as well: genetic algorithm [1], genetic programming [2], and evolutionary programming [3, 4].

On the other hand, local search (LS) algorithms are single-solution based metaheuristics, which show their efficiency in tackling many optimization problems in different domains [5]. LS methods move from one solution to another in the search space of candidate solutions until a nearly-optimal solution is found. Although the use of LS algorithms allows to reduce the computational complexity of the search process, optimization problems are often NP-hard and objective function calculations on CPU remain time-consuming, especially when the size of the search space is huge. As a consequence, parallelism seems necessary to speed up the search, improve the quality of the obtained solutions, improve the robustness and solve large scale problems. Re-think the design and implementation of parallel local searches on GPU architecture might be considered in an efficient way.

The aim of this paper is to present a new general methodology for constructing local search methods on GPU. To the best of our knowledge, approaches on local search algorithms have never been proposed in the literature. This novel approach is based on the idea that parallelism can be done at iteration-level: each solution of a given neighbor can be evaluated in parallel. The LS iterative process is managed by the CPU and the evaluation function is called on GPU. This way, this approach tends to be generic for many local search algorithms.

The organization of the paper is the following. In the next section, NVIDIA GPUs characteristics are described. Section 3 presents a general approach for designing LS methods on GPU. In Section 4, an implementation of LS algorithms is depicted according to the CUDA model. For testing performances of this approach, three NP-complete problems have been implemented on GPU and results will be discussed in Section 5. An extension of the general approach is made for multiple LS methods on multi-GPU in section 6. Finally, conclusions and perspectives of this work are drawn in Section 7.

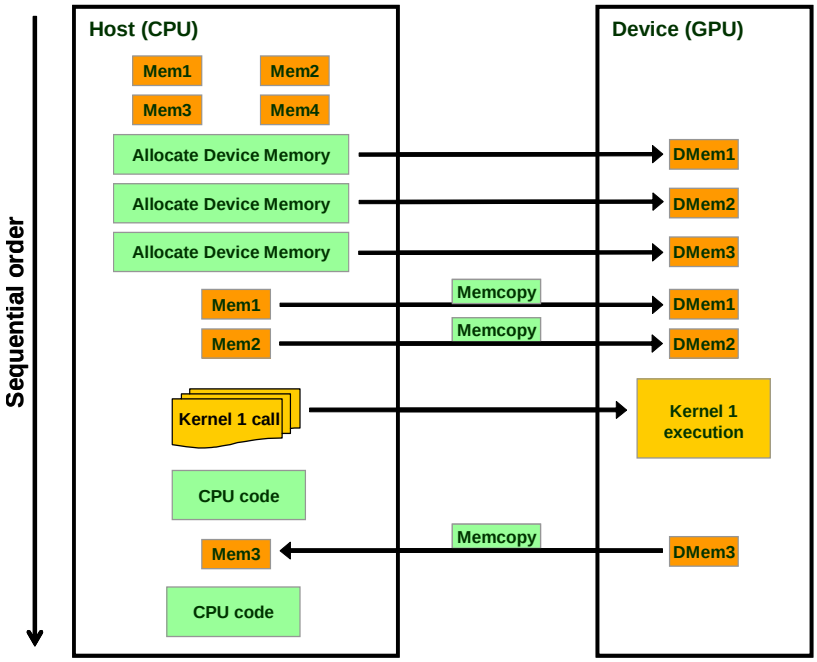

Figure 1: Illustration general GPU model

2 Graphics Processing Unit

2.1 General GPU Model

In general-purpose computing on graphics process units, CPU is considered as a host and the GPU is exposed as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer, where data must be transferred between the memory space of the host and device.

Each device processor supports the single program multiple data (SPMD) model, i.e. multiple autonomous processors simultaneously execute the same program on different data. For achieving this, the notion of kernel is defined. It is a function callable from the host and executed on the specified device simultaneously by several processors in parallel.

Figure 1 illustrates an example of this concept. One can notice that kernel calls are asynchronous. After a kernel launch, control immediately returns to the host CPU. It may be seen as an efficient way to overlap computation on the host and device. Memory transfer from the CPU to the device memory is a synchronous operation which is time consuming. Bus bandwith and latency between CPU and GPU can significantly decrease performance of a program. As a consequence, one has to minimize these memory copies.

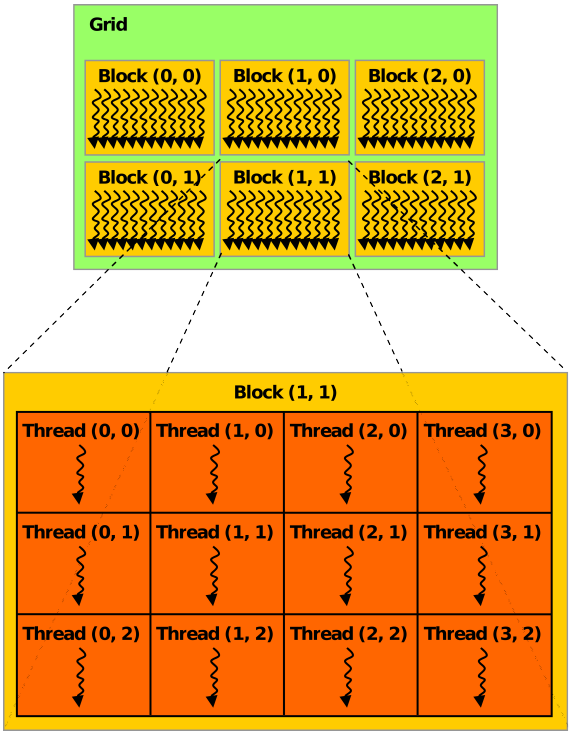

Figure 2: CUDA threads model (source from CUDA programming guide [6])

2.2 CUDA Threading Model

CUDA (Compute Unified Device Architecture) is a parallel computing environment, which provides an application programming interface for NVIDIA architectures [6]. The notion of thread in CUDA doesn't have exactly the same meaning as CPU thread. A thread on GPU is an element of the data to be processed. Compared to CPU threads, CUDA threads are lightweight [7]. That means that changing the context between two threads is not a costly operation.

Threads are organized within so called thread blocks. A kernel is executed by multiple equally thread blocks. Figure 2 illustrates these multiple blocks organization. Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are regrouped in a similar manner. First, the advantage of grouping is that the number of blocks processed simultaneously by the GPU are closely linked to hardware resources. Secondly, since each thread is provided with a unique *id* that can be used to compute on different data, this model of threads provides an easy abstraction for SIMD architecture.

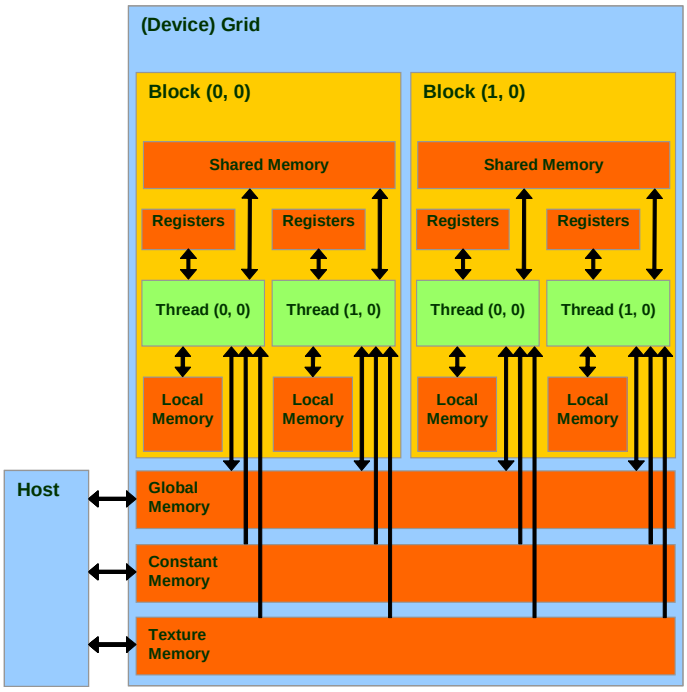

Figure 3: CUDA memory model (source from CUDA programming guide [6])

2.3 Memory Management

From a hardware point of view, graphics cards consist of streaming multiprocessors, each with processing units, registers and on-chip memory. Since multiprocessors work according to the SPMD model, they share the same code and have access to different memory areas. Figure 3 illustrates these different available memories and connections with thread blocks.

Communication between CPU host and its device is done through global memory. Since this memory is not cached and access is slow, one needs to minimize accesses to global memory and reuse data within the local multiprocessor memories. Graphic cards provide also read only texture memory to accelerate operations such as 2D or 3D mapping. Texture units are provided to make graphic operations occur relatively fast. Constant memory is read only from kernels and is hardware optimized for the case where all threads read the same location. Shared memory is a fast memory located on the multiprocessors and shared by threads of each thread block. This memory area provides a way for threads to communicate within the same block. Registers among streaming processors are partitioned among the threads running on it, they constitute fast memory access. Local memory is a memory abstraction and is not an actual hardware component. In fact, local memory resides in global memory allocated by the compiler.

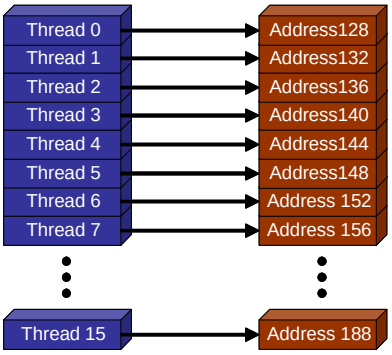

Figure 4: An example of coalesced global memory access pattern

2.4 Memory Coalescing

Each block of threads is split into SIMD groups of threads called warps. At any clock cycles, each processor of the multiprocessor selects a warp that is ready to execute the same instruction on different data. For being efficient, global memory accesses must be coalesced, which means that a memory read by consecutive threads in a warp is combined by the hardware into several memory reads. The requirement is that threads of the same warp must read global memory in an order manner (Fig. 4). Global memory accesses patterns that are non-coalesced can significantly decrease the performances of a program.

3 Designing Local Search Algorithms on GPU

LS algorithms are a class of metaheuristics which have been used successfully for many optimization problems. A LS algorithm begins with an initial solution and then iteratively improves that solution by moving to a neighbor solution. When incremental evaluation is possible, it provides efficient speed-up. Nevertheless, even with this technique, LS methods require long computational time on very large instances. As a consequence, parallelization is necessary to reduce computational time for large scale instances. Figure 5 gives a general model for constructing a LS algorithm.

3.1 General model

LS profiling shows that the most resource-consuming part of a LS is the evaluation of the generated solutions. As a result, the generation and evaluation of the neighborhood might be done in parallel in deterministic LS (e.g. hill climbing, tabu search, variable neighborhood search). In this parallel model, the neighborhood is divided into different partitions which are evaluated in a parallel independant way.

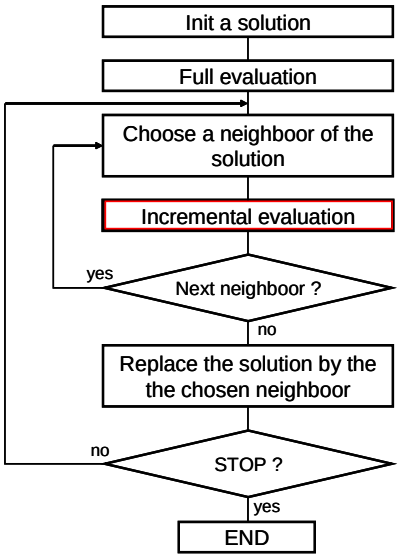

Figure 5: General model for local search algorithms

Regarding GPUs, graphics cards have evolved into a highly parallel and multithreaded environment. By definition a GPU is organized following the SMPD model, meaning that multiple autonomous processors simultaneously execute the same program at independent points.

Therefore, the mapping between the LS parallel model and the GPU model is quiet straightforward. Let us divide the neighborhood into m elements (partitions of size equals to one) where m is the size of the neighborhood. This way, one candidate solution is represented by one GPU process. An extended LS construction model on GPU is proposed in Fig. 6.

This model can be seen as a cooperative model between the CPU and the GPU. Indeed, the GPU is used as a coprocessor in a synchronous manner. The resource-consuming part i.e. the incremental evaluation, is calculated by the GPU and the rest is handled by the CPU. From an hardware implementation point of view, GPUs are efficient with high intensive arithmetic instructions, and CPUs in complex instructions.

3.2 The Proposed GPU Algorithm

Adapting traditional LS methods on GPU is not a straighforward task because memory management on GPU have to be handled. As previously said, memory transferts from CPU on GPU are slow and these copies have to be minimized. Algorithm 1 proposes a methodology to adapt LS methods on GPU in a generic way. This methodology fits well with the previous general GPU model example mentioned in Fig. 1. First of all, at initialization stage, memory allocations on device GPU have to be made. Inputs of the problem and candidate solution

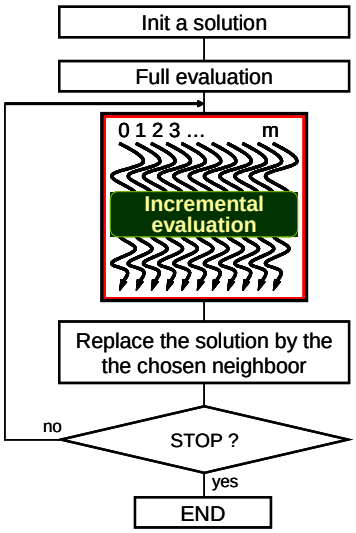

Figure 6: General model for local search algorithm on GPU

Algorithm 1 Local Search Template on GPU

1: Choose an initial solution
2: Evaluate the solution
3: Specific LS initializations
4: Allocate problem data inputs on GPU device memory
5: Allocate a solution on GPU device memory
6: Allocate a neighborhood fitnesses structure on GPU device memory
7: Allocate additional solution structures on GPU device memory
8: Copy problem data inputs on GPU device memory
9: Copy the solution on GPU device memory
10: Copy additional solution structures on GPU device memory
11: **repeat**
12: **for** each neighbor in parallel **do**
13: Incremental evaluation of the candidate solution
14: Insert the resulting fitness into the neighborhood fitnesses structure
15: **end for**
16: Copy neighborhood fitnesses structure on CPU host memory
17: Specific LS solution selection strategy on the neighborhood fitnesses structure
18: Specific LS post-treatment
19: Copy the chosen solution on GPU device memory
20: Copy additional solution structures on GPU device memory
21: **until** a stopping criteria satisfied

must be allocated (lines 4 and 5). Since GPUs require massively computations with predictable memory accesses, a structure has to be allocated for stocking all the neighborhood fitnesses at different addresses (line 6). Additional solution structures which are problem-dependent can also be allocated to facilitate the computation of incremental evaluation (line 7). Secondly, problem data inputs, initial candidate solution and additional structures associated to this solution have to be copied on the GPU (lines 8 to 10). Notice that problem inputs are copied only once during the process. Third, comes the parallel iteration-level, in which each neighbor solution is evaluated and copied in the neighborhood fitnesses structure (from lines 12 to 15). The order in which candidate neighbors are evaluated is undefined. Fourth, the neighborhood fitnesses structure has to be copied to the host CPU (line 16). Then a specific LS solution selection strategy is applied to this structure (line 17): the exploration of the neighborhood fitnesses structure is done by the CPU in a sequential way. And finally, after a new candidate has been chosen, this latter and its additional structures are copied to the GPU (lines 19 and 20). The process is repeated until a terminal condition.

Since the GPU parallelization part is common to both LS methods, this algorithm can be easily generalized for advanced LS methods such as iterated local search or variable neighborhood search.

4 Implementing Local Search Algorithms on GPU

For LS procedures, neighborhood structures are the main part of those algorithms. These structures play a crucial role in the performance of LS methods and are problem-dependent. As previously seen in Fig. 2, CUDA model works with thread blocks. A kernel (parallel function) is launched with a large number of threads, which are provided with a unique *id*. As a consequence, the main difficulty which remains, is to find a correct mapping between a LS neighbor candidate solution and a GPU thread.

Neighborhood structure strongly depends on the target optimization problem representation. Three major encodings in the literature can be highlighted: Binary encoding (e.g. Knapsack, SAT), vector of discrete values (e.g. location problem, assignment problem) and permutation (e.g. TSP, scheduling problems).

4.1 Binary Representation

Neighborhood representation for binary problems is based on Hamming distance. The neighborhood of a given solution consists in flipping one bit of the solution (for a Hamming distance of one).

A mapping between LS neighborhood encoding and GPU threads is quiet trivial. Indeed, on one side, for a binary vector of size n, the size of the neighborhood is exactly n. On the other hand, threads are provided with a unique *id*. That way, the incremental evaluation kernel is launched with n threads (a

neighbor is associated with a single thread), and the size of the neighborhood fitnesses structure allocated on GPU is n. As a result, a $\mathbb{N} \rightarrow \mathbb{N}$ mapping can be made.

4.2 Discrete Vector Representation

Discrete vector representation is an extension of binary encodings using a given alphabet Σ. In this way, discrete value of a vector element is replaced by any other character of the alphabet.

Let us consider that the cardinality of the alphabet Σ is k, the size of the neighborhood will be $(k-1) \times n$ for a discret vector of size n. In a similar manner, a $\mathbb{N} \rightarrow \mathbb{N}$ mapping here can also be made. $(k-1) \times n$ threads execute the incremental evaluation kernel, and a neighborhood fitnesses structure of size $(k-1) \times n$ has to be provided.

4.3 Permutation Representation

Neighborhood built by pairwise exchanging operations (known as the 2-exchange neighborhood) is a standard way for permutation problems. For a permutation of size n, the size of this neighborhood is $\frac{n \times (n-1)}{2}$.

Like previous representations, the incremental evaluation kernel is executed by $\frac{n \times (n-1)}{2}$ threads, and the size of neighborhood fitnesses structure is $\frac{n \times (n-1)}{2}$. However, for permutation encoding a mapping between a neighbor and a GPU thread is not straightforward. Indeed, on the one hand, a neighbor is composed by two element indexes (a swap in a permutation). On the other hand, threads are identified by a unique *id*. As a result, $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ and $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ bijections have to be considered.

Let us consider a 2D abstraction. In this latter, elements of the neighborhood are disposed in a zero-based indexing 2D representation in a similar way that a lower triangular matrix. Let n be the size of the solution representation and let $m = \frac{n \times (n-1)}{2}$ be the size of the neighborhood. Let i and j be the indexes of two elements to exchange in a permutation. A candidate neighbor is then identified by both i and j indexes in the 2D abstraction. Let $f(i,j)$ be the corresponding index in the 1D neighborhood fitnesses structure. Figure 7 gives an illustration of this abstraction.

In this example, $n = 6, m = 14$ and the neighbor identified by the coordinate $(i = 2\ ;\ j = 3)$ is mapped to the corresponding 1D array element $f(i,j) = 9$.

4.3.1 $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ Bijection:

The neighboor represented by the $(i\ ;\ j)$ coordinates is known, and its corresponding index $f(i,j)$ on the 1D structure has to be calculated. In a matrix approach, if the 1D array size was $n * n$, the $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping would be:

$$f(i,j) = i \times (n-1) + (j-1)$$

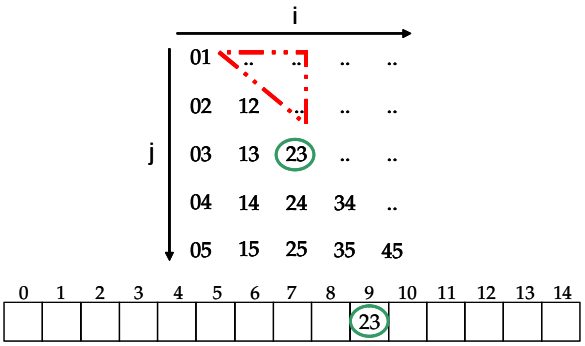

Figure 7: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ bijection

Since the 1D array size m is $\frac{n \times (n-1)}{2}$, in the 2D abstraction, elements above the diagonal preceding the neighbor must not be considered (illustrated in Fig. 7 by a triangle). The corresponding mapping is therefore:

$$f(i,j) = i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2} \tag{1}$$

$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ bijection is done.

4.3.2 $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ Bijection:

$f(i,j)$ is a given index of the neighborhood fitnesses array, and i and j have to be found.

If the element corresponding to $f(i,j)$ in the 2D abstraction has a given i abscissa, then let k be the distance between the $i+1$ and $n-2$ abscissas. If k is known, the value of i can be deduce:

$$i = n - 2 - k \tag{2}$$

Let X be the number of elements following $f(i,j)$ in the neighborhood index-based array numerotation:

$$X = m - f(i,j) - 1 \tag{3}$$

Since this number can be also represented in the 2D abstraction, the main idea is to maximize the distance k such as:

$$\frac{k \times (k+1)}{2} \leq X \tag{4}$$

Figure 8 gives an illustration of this idea (represented by a triangle).

Resolving (4) gives the greatest distance k:

$$k \times (k+1) \leq 2X$$

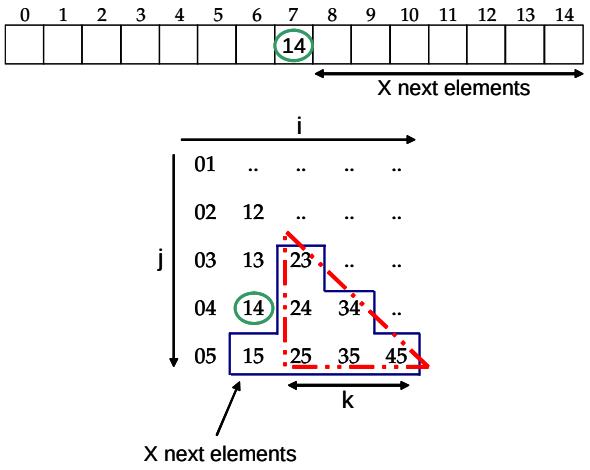

Figure 8: $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ bijection

$$(k + \frac{1}{2})^2 - \frac{1}{4} \leq 2X$$
$$k + \frac{1}{2} \leq \sqrt{2X + \frac{1}{4}}$$
$$k \leq \frac{\sqrt{8X+1} - 1}{2}$$
$$k = \lfloor \frac{\sqrt{8X+1} - 1}{2} \rfloor \quad (5)$$

A value of i can then be calculated according to (2). Finally, by using (1) j can be given by:

$$j = f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1 \quad (6)$$

Since $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ bijection is also done, a mapping between a neighbor and a GPU thread can be made. Notice that for binary-problem encodings, the mapping for neighborhood based on Hamming distance of two can be done in a similar manner.

From an implementation point of view, some operations on some GPU architectures suffer from simple and double float precision. For example, according to NVIDIA programming guide, CUDA square root library function has a maximum ulp error of 3. For example, square root precision on GPU doesn't have the same precision on CPU. This problem can be fixed by adding a small value ϵ to the square root argument during implementation.

5 Experiments

5.1 Configuration

To evaluate the performances of LS implementation, three problems in combinatorial optimization have been implemented on GPU on top of three different configurations. Configuration one is given by a Core 2 Duo 2Ghz and a GeForce 8600M GT with 4 multiprocessors. The second one is composed by a Core 2 Quad 2.4Ghz and a GeForce 8800 GTX with 16 multiprocessors. The third configuration is an Intel Xeon 3Ghz with a GeForce GTX 280 with 32 multiprocessors.

For the three problems, an iterated tabu search has been implemented. The number of local iterations of the tabu search is 1000, and the number of global iterations for the iterative process is 10. The experiments intend to measure the speed-up and not the quality of solutions. Indeed, for each problem, there are some better well-suited specific LS methods. That is the reason why for the next results, time measurement and acceleration factor (compared to a single CPU) are only represented. For each instance of a problem, a standalone CPU implementation, a CPU-GPU, and a CPU-GPU version using texture memory are measured for each configuration (average time for 10 executions).

5.2 Memory Coalescing and Texture Memory

Concerning global memory, if a memory transaction cannot be coalesced, then a separate memory transaction will be issued for each thread in the warp (active group of threads). The performance penalty for non-coalesced memory accesses varies according to the size of the data structure.

For LS methods, and especially for permutation problems, memory coalescing is difficult to realize. As a result, it can lead to a significantly performance decrease.

A solution for this drawback is to use texture memory. Indeed, this latter is an alternative memory access path that can be bound to regions of the global memory. Each texture unit has some internal memory that buffers data from global memory. Therefore, texture memory can be seen as a relaxed mechanism for the thread processors to access global memory because the coalescing requirements do not apply to texture memory accesses. Since data accesses are frequent in LS incremental evaluation methods, using texture memory can provide a large performance increase. Optimization problem inputs such as 2D matrices or 1D vector solution can be bound to texture memory.

Notice that in the 200-series, global memory is easier to access due to the relaxation of the coalescing rules. It means that applications developped in CUDA get better global memory performance.

Table 1: Quadratic assignment problem pairwise-exchange neighborhood

Instance	Configuration 1			Configuration 2			Configuration 3		
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}
tai12a	0.1	$1.1_{\times 0.1}$	$0.8_{\times 0.2}$	0.1	$0.6_{\times 0.1}$	$0.5_{\times 0.1}$	0.1	$0.5_{\times 0.2}$	$0.5_{\times 0.2}$
tai15a	0.3	$1.4_{\times 0.2}$	$0.9_{\times 0.3}$	0.1	$0.8_{\times 0.2}$	$0.6_{\times 0.2}$	0.2	$0.6_{\times 0.3}$	$0.6_{\times 0.3}$
tai17a	0.4	$1.6_{\times 0.2}$	$1.0_{\times 0.4}$	0.2	$0.9_{\times 0.2}$	$0.6_{\times 0.3}$	0.3	$0.7_{\times 0.4}$	$0.6_{\times 0.5}$
tai20a	0.6	$2.1_{\times 0.3}$	$1.0_{\times 0.6}$	0.3	$1.2_{\times 0.2}$	$0.7_{\times 0.4}$	0.4	$0.8_{\times 0.5}$	$0.7_{\times 0.7}$
tai25a	1.3	$2.8_{\times 0.5}$	$1.2_{\times \mathbf{1.1}}$	0.5	$1.9_{\times 0.3}$	$0.9_{\times 0.6}$	0.8	$1.0_{\times 0.9}$	$0.7_{\times \mathbf{1.1}}$
tai30a	2.2	$4.3_{\times 0.5}$	$1.3_{\times \mathbf{1.7}}$	0.9	$2.2_{\times 0.4}$	$1.0_{\times 0.9}$	1.5	$1.1_{\times \mathbf{1.3}}$	$0.8_{\times \mathbf{1.8}}$
tai35a	3.4	$6.6_{\times 0.5}$	$1.5_{\times \mathbf{2.3}}$	1.4	$2.9_{\times 0.5}$	$1.2_{\times \mathbf{1.2}}$	2.3	$1.2_{\times \mathbf{1.9}}$	$0.9_{\times \mathbf{2.5}}$
tai40a	4.9	$9.8_{\times 0.5}$	$1.9_{\times \mathbf{2.5}}$	2.1	$3.8_{\times 0.6}$	$1.4_{\times \mathbf{1.5}}$	3.5	$1.4_{\times \mathbf{2.6}}$	$1.1_{\times \mathbf{3.3}}$
tai50a	9.7	$18_{\times 0.5}$	$3.1_{\times \mathbf{3.1}}$	4.1	$5.7_{\times 0.7}$	$1.7_{\times \mathbf{2.4}}$	6.8	$1.7_{\times \mathbf{4.1}}$	$1.3_{\times \mathbf{5.3}}$
tai60a	16	$30_{\times 0.6}$	$4.9_{\times \mathbf{3.4}}$	7.0	$8.5_{\times 0.8}$	$1.9_{\times \mathbf{3.6}}$	11	$2.0_{\times \mathbf{6.0}}$	$1.6_{\times \mathbf{7.7}}$
tai64c	27	$37_{\times 0.7}$	$5.3_{\times \mathbf{5.2}}$	8.7	$10_{\times 0.8}$	$2.1_{\times \mathbf{4.1}}$	14	$2.1_{\times \mathbf{6.8}}$	$1.6_{\times \mathbf{8.9}}$
tai80a	43	$72_{\times 0.6}$	$10_{\times \mathbf{4.2}}$	18	$20_{\times 0.9}$	$4.6_{\times \mathbf{4.0}}$	29	$3.3_{\times \mathbf{9.0}}$	$2.7_{\times \mathbf{10.9}}$
tai100a	111	$143_{\times 0.8}$	$20_{\times \mathbf{5.5}}$	38	$37_{\times \mathbf{1.0}}$	$9.6_{\times \mathbf{4.0}}$	61	$5.5_{\times \mathbf{11.1}}$	$3.5_{\times \mathbf{17.5}}$
tai150b	349	$521_{\times 0.7}$	$86_{\times \mathbf{4.0}}$	158	$108_{\times \mathbf{1.5}}$	$36_{\times \mathbf{4.4}}$	228	$18_{\times \mathbf{12.6}}$	$12_{\times \mathbf{18.8}}$
tai256c	1879	$2959_{\times 0.6}$	$824_{\times \mathbf{2.3}}$	899	$467_{\times \mathbf{1.9}}$	$189_{\times \mathbf{4.7}}$	1186	$158_{\times \mathbf{7.5}}$	$56_{\times \mathbf{20.8}}$

5.3 Quadratic Assignment Problem

The quadratic assignment problem (QAP) arises in many applications such as facility location or data analysis. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ matrices of positive integers. Then finding a solution of the QAP is equivalent to find a permutation $\pi = (1, 2, \ldots, n)$ that minimizes the objective function:

$$z(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i)\pi(j)}$$

The incremental evaluation function has a time complexity of $O(n)$, and the number of threads executed is equals to $\frac{n \times (n-1)}{2}$. Results are shown in Table 1 for the three configurations. Time measurement is represented in seconds, and for both GPU implementation and GPU version using texture memory, acceleration factors compared to a standalone CPU are represented in subindexes. Due to high misaligned accesses to global memories (flows and distances in QAP), memory non-coalescing reduces seriously the performance of the GPU implementation on both G80 cards. Binding texture on global memory can overcome the problem. Indeed, from the instance tai35a, using texture memory starts giving positive acceleration factors for both configurations (from $\times 1.2$ to $\times 2.5$). GPU keeps accelerating the LS process as long as the size grows. Concerning GTX 280, this card provides twice more multiprocessors and registers. As a consequence, hardware capability and coalescing rules relaxation lead to a very significant speed-up (from $\times 7.5$ to $\times 20.8$ for the biggest instance tai256c) .

5.4 Permuted Perceptron Problem

In [8], Pointcheval introduced a cryptographic identification scheme based on the perceptron problem, which seems to be suited for resource constrained devices such as smart cards. An ϵ-vector is a vector with all entries being either +1 or -1. Similarly an ϵ-matrix is a matrix in which all entries are either +1 or -1. The permuted perceptron problem (PPP) is as follows:

Definition Given an ϵ-matrix A of size $m \times n$, find an ϵ-vector V of size n such that $AV \geq 0$

The permuted perceptron problem (PPP) is a harder variant of the PP:

Definition Given an ϵ-matrix A of size $m \times n$ and a multiset S of non-negative integers of size m, find an ϵ-vector V of size n such that $\{\{(AV)_j / j = \{1, \ldots, m\}\}\} = S$.

PPP has been implemented using a binary encoding. Part of the full evaluation of a solution can be seen as a matrix-vector product. Then the incremental evaluation of a neighbor can be done in linear time. Results for a Hamming neighborhood of distance one is depicted in Table 2 (m-n instances). From $m = 301$ and $n = 317$, GPU version using texture memory starts to be faster than CPU version for both configurations (from $\times 1.4$ to $\times 1.6$). Since accesses to global memory in the incremental evaluation are minimized, GPU implementation is not much affected from non-coalescing memory operations. Indeed, from $m = 601$ and $n = 617$, GPU version without any texture memory use starts to give better results (from $\times 1.1$ to $\times 2.2$). The speed-up grows as long as the size increase (up to $\times 8$ for $m = 1301$, $n = 1317$).

Acceleration factor for this implementation is significant but not huge. This can be explained by the fact that since the neighborhood is relatively small (n threads), the number of threads per block is not enough to fully cover the memory access latency.

For confirming this point, Hamming neighborhood of distance two on GPU has also been implemented. Incremental evaluation is executed by $\frac{n \times (n-1)}{2}$ threads. Then the results figure in Table 3. For the first instance ($m = 73$, $n = 73$), acceleration factors of the texture version are already important (from $\times 3.6$ to $\times 10.9$). As long as the instance size grows, acceleration factor becomes efficient (from $\times 3.6$ to $\times 8.1$ for the first configuration). Since a large number multiprocessors are available on both 8800 and GTX 280, high speed-ups can be highlighted (from $\times 9.6$ $\times 42.6$). As a consequence, parallelization on GPU provides an efficient way for handling large neighborhoods.

5.5 Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the most popular combinatorial optimization problems. Given n cities and a distance matrix $d_{n,n}$, where each element d_{ij} represents the distance between the cities i and j, find a tour which

Table 2: Permuted perceptron problem 1-Hamming distance neighborhood

Instance	Configuration 1			Configuration 2			Configuration 3		
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}
73-73	1.5	$6.2_{\times 0.2}$	$5.0_{\times 0.3}$	1.1	$3.4_{\times 0.3}$	$3.0_{\times 0.4}$	1.1	$3.5_{\times 0.3}$	$3.0_{\times 0.4}$
81-81	1.9	$6.8_{\times 0.3}$	$5.3_{\times 0.3}$	1.4	$3.8_{\times 0.4}$	$3.4_{\times 0.4}$	1.3	$3.8_{\times 0.3}$	$3.3_{\times 0.4}$
101-101	2.8	$8.3_{\times 0.3}$	$6.4_{\times 0.4}$	2.1	$4.5_{\times 0.5}$	$4.1_{\times 0.5}$	2.0	$4.5_{\times 0.4}$	$4.0_{\times 0.5}$
101-117	3.3	$8.9_{\times 0.4}$	$6.6_{\times 0.5}$	2.5	$4.8_{\times 0.5}$	$4.3_{\times 0.6}$	2.2	$4.9_{\times 0.4}$	$4.2_{\times 0.5}$
201-217	11	$20_{\times 0.6}$	$12_{\times 0.9}$	8.8	$10_{\times 0.8}$	$8.1_{\times \mathbf{1.1}}$	8.1	$8.8_{\times 0.9}$	$7.7_{\times \mathbf{1.1}}$
301-317	24	$34_{\times 0.7}$	$18_{\times \mathbf{1.4}}$	19	$16_{\times \mathbf{1.1}}$	$13_{\times \mathbf{1.5}}$	17	$12_{\times \mathbf{1.4}}$	$11_{\times \mathbf{1.6}}$
401-417	43	$54_{\times 0.8}$	$24_{\times \mathbf{1.7}}$	34	$28_{\times \mathbf{1.2}}$	$20_{\times \mathbf{1.7}}$	31	$16_{\times \mathbf{1.9}}$	$14_{\times \mathbf{2.2}}$
501-517	116	$140_{\times 0.8}$	$99_{\times \mathbf{1.2}}$	65	$52_{\times \mathbf{1.3}}$	$42_{\times \mathbf{1.6}}$	55	$43_{\times \mathbf{1.3}}$	$40_{\times \mathbf{1.4}}$
601-617	189	$169_{\times \mathbf{1.1}}$	$98_{\times \mathbf{1.9}}$	134	$96_{\times \mathbf{1.4}}$	$77_{\times \mathbf{1.7}}$	105	$47_{\times \mathbf{2.2}}$	$43_{\times \mathbf{2.4}}$
701-717	288	$198_{\times \mathbf{1.4}}$	$126_{\times \mathbf{2.3}}$	202	$119_{\times \mathbf{1.7}}$	$98_{\times \mathbf{2.1}}$	148	$51_{\times \mathbf{2.9}}$	$47_{\times \mathbf{3.1}}$
801-817	375	$248_{\times \mathbf{1.5}}$	$122_{\times \mathbf{3.1}}$	269	$125_{\times \mathbf{2.1}}$	$100_{\times \mathbf{2.7}}$	200	$55_{\times \mathbf{3.6}}$	$50_{\times \mathbf{4.0}}$
901-917	499	$287_{\times \mathbf{1.7}}$	$134_{\times \mathbf{3.7}}$	337	$132_{\times \mathbf{2.5}}$	$103_{\times \mathbf{3.2}}$	262	$59_{\times \mathbf{4.4}}$	$54_{\times \mathbf{4.8}}$
1001-1017	596	$348_{\times \mathbf{1.7}}$	$146_{\times \mathbf{4.1}}$	436	$145_{\times \mathbf{3.0}}$	$107_{\times \mathbf{4.0}}$	336	$63_{\times \mathbf{5.3}}$	$58_{\times \mathbf{5.8}}$
1101-1117	698	$430_{\times \mathbf{1.6}}$	$200_{\times \mathbf{3.5}}$	546	$210_{\times \mathbf{2.6}}$	$173_{\times \mathbf{3.2}}$	427	$85_{\times \mathbf{5.0}}$	$78_{\times \mathbf{5.4}}$
1201-1217	981	$510_{\times \mathbf{1.9}}$	$241_{\times \mathbf{4.1}}$	658	$225_{\times \mathbf{2.9}}$	$177_{\times \mathbf{3.7}}$	539	$88_{\times \mathbf{6.1}}$	$82_{\times \mathbf{6.6}}$
1301-1317	1176	$573_{\times \mathbf{2.1}}$	$288_{\times \mathbf{4.1}}$	784	$228_{\times \mathbf{3.4}}$	$180_{\times \mathbf{4.3}}$	687	$93_{\times \mathbf{7.4}}$	$85_{\times \mathbf{8.0}}$

Table 3: Permuted perceptron problem 2-Hamming distance neighborhood

Instance	Configuration 1			Configuration 2			Configuration 3		
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}
73-73	2.9	$3.9_{\times 0.7}$	$0.8_{\times \mathbf{3.6}}$	2.2	$1.3_{\times \mathbf{1.7}}$	$0.2_{\times \mathbf{9.6}}$	2.1	$0.2_{\times \mathbf{9.9}}$	$0.2_{\times \mathbf{10.9}}$
81-81	3.9	$5.2_{\times 0.8}$	$1.0_{\times \mathbf{4.0}}$	2.8	$1.7_{\times \mathbf{1.7}}$	$0.3_{\times \mathbf{10.4}}$	2.7	$0.3_{\times \mathbf{10.3}}$	$0.2_{\times \mathbf{12.2}}$
101-101	7.6	$9.7_{\times 0.8}$	$1.7_{\times \mathbf{4.4}}$	5.1	$2.9_{\times \mathbf{1.7}}$	$0.4_{\times \mathbf{12.0}}$	5.2	$0.4_{\times \mathbf{12.8}}$	$0.3_{\times \mathbf{18.1}}$
101-117	10	$13_{\times 0.7}$	$2.5_{\times \mathbf{4.0}}$	7.3	$4.1_{\times \mathbf{1.8}}$	$0.6_{\times \mathbf{13.0}}$	7.0	$0.6_{\times \mathbf{12.8}}$	$0.4_{\times \mathbf{19.0}}$
201-217	68	$82_{\times 0.8}$	$15_{\times \mathbf{4.3}}$	51	$25_{\times \mathbf{2.0}}$	$3.3_{\times \mathbf{15.3}}$	49	$3.0_{\times \mathbf{16.4}}$	$1.9_{\times \mathbf{25.7}}$
301-317	232	$251_{\times 0.9}$	$58_{\times \mathbf{4.0}}$	174	$61_{\times \mathbf{2.8}}$	$10_{\times \mathbf{16.0}}$	169	$9.5_{\times \mathbf{17.8}}$	$6.2_{\times \mathbf{27.4}}$
401-417	573	$570_{\times \mathbf{1.0}}$	$147_{\times \mathbf{3.9}}$	443	$125_{\times \mathbf{3.5}}$	$24_{\times \mathbf{17.8}}$	405	$21_{\times \mathbf{19.3}}$	$14_{\times \mathbf{28.6}}$
501-517	1235	$1105_{\times \mathbf{1.1}}$	$294_{\times \mathbf{4.2}}$	901	$220_{\times \mathbf{4.1}}$	$51_{\times \mathbf{17.4}}$	804	$40_{\times \mathbf{19.7}}$	$29_{\times \mathbf{27.5}}$
601-617	3208	$1881_{\times \mathbf{1.7}}$	$512_{\times \mathbf{6.3}}$	2526	$355_{\times \mathbf{7.1}}$	$88_{\times \mathbf{28.5}}$	2056	$67_{\times \mathbf{30.5}}$	$51_{\times \mathbf{40.2}}$
701-717	5592	$3002_{\times \mathbf{1.9}}$	$824_{\times \mathbf{6.8}}$	4538	$546_{\times \mathbf{8.3}}$	$142_{\times \mathbf{31.9}}$	3590	$105_{\times \mathbf{34.2}}$	$84_{\times \mathbf{42.3}}$
801-817	8611	$4396_{\times \mathbf{2.0}}$	$1245_{\times \mathbf{6.9}}$	6922	$815_{\times \mathbf{8.5}}$	$210_{\times \mathbf{32.9}}$	5405	$152_{\times \mathbf{35.4}}$	$128_{\times \mathbf{42.2}}$
901-917	12398	$6245_{\times \mathbf{2.0}}$	$1788_{\times \mathbf{6.9}}$	9952	$1088_{\times \mathbf{9.1}}$	$300_{\times \mathbf{33.2}}$	7900	$215_{\times \mathbf{36.7}}$	$187_{\times \mathbf{42.1}}$
1001-1017	17507	$8474_{\times \mathbf{2.1}}$	$2502_{\times \mathbf{7.0}}$	14445	$1469_{\times \mathbf{9.8}}$	$416_{\times \mathbf{34.7}}$	11083	$291_{\times \mathbf{38.1}}$	$262_{\times \mathbf{42.2}}$
1101-1117	26079	$11175_{\times \mathbf{2.3}}$	$3336_{\times \mathbf{7.8}}$	19227	$1878_{\times \mathbf{10.2}}$	$551_{\times \mathbf{34.9}}$	14974	$401_{\times \mathbf{37.3}}$	$357_{\times \mathbf{41.9}}$
1201-1217	34553	$14108_{\times \mathbf{2.4}}$	$4348_{\times \mathbf{7.9}}$	25664	$2407_{\times \mathbf{10.7}}$	$705_{\times \mathbf{36.4}}$	19532	$512_{\times \mathbf{38.1}}$	$460_{\times \mathbf{42.4}}$
1301-1317	39484	$17910_{\times \mathbf{2.2}}$	$4903_{\times \mathbf{8.1}}$	33170	$3050_{\times \mathbf{10.9}}$	$912_{\times \mathbf{36.4}}$	25026	$647_{\times \mathbf{38.7}}$	$587_{\times \mathbf{42.6}}$

Table 4: Traveling salesman problem pairwise-exchange neighborhood

Instance	Configuration 1			Configuration 2			Configuration 3		
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}
eil101	2.3	$3.7_{\times 0.6}$	$1.8_{\times \mathbf{1.3}}$	1.7	$1.8_{\times 1.0}$	$1.0_{\times \mathbf{1.7}}$	1.4	$0.6_{\times \mathbf{2.3}}$	$0.6_{\times \mathbf{2.4}}$
d198	10	$13_{\times 0.8}$	$7.1_{\times \mathbf{1.4}}$	7.3	$5.4_{\times \mathbf{1.3}}$	$3.2_{\times \mathbf{2.2}}$	5.6	$1.5_{\times \mathbf{3.7}}$	$1.4_{\times \mathbf{4.0}}$
lin318	27	$39_{\times 0.7}$	$19_{\times \mathbf{1.4}}$	14	$13_{\times \mathbf{1.1}}$	$8.1_{\times \mathbf{1.8}}$	14	$3.5_{\times \mathbf{4.1}}$	$3.3_{\times \mathbf{4.4}}$
rd400	33	$65_{\times 0.5}$	$26_{\times \mathbf{1.3}}$	23	$19_{\times \mathbf{1.2}}$	$12_{\times \mathbf{1.8}}$	23	$5.8_{\times \mathbf{4.1}}$	$5.6_{\times \mathbf{4.2}}$
pcb442	57	$89_{\times 0.6}$	$37_{\times \mathbf{1.5}}$	29	$23_{\times \mathbf{1.3}}$	$14_{\times \mathbf{2.1}}$	29	$7.0_{\times \mathbf{4.2}}$	$6.5_{\times \mathbf{4.5}}$
att532	63	$137_{\times 0.5}$	$59_{\times \mathbf{1.1}}$	44	$39_{\times \mathbf{1.1}}$	$23_{\times \mathbf{1.9}}$	40	$10_{\times \mathbf{3.9}}$	$9.8_{\times \mathbf{4.2}}$
rat783	196	$321_{\times 0.6}$	$152_{\times \mathbf{1.3}}$	107	$78_{\times \mathbf{1.4}}$	$47_{\times \mathbf{2.3}}$	93	$26_{\times \mathbf{3.5}}$	$23_{\times \mathbf{4.0}}$
pcb1173	583	$744_{\times 0.8}$	$391_{\times \mathbf{1.5}}$	333	$181_{\times \mathbf{1.8}}$	$106_{\times \mathbf{3.1}}$	270	$67_{\times \mathbf{4.0}}$	$57_{\times \mathbf{4.7}}$
d1291	692	$892_{\times 0.8}$	$509_{\times \mathbf{1.4}}$	492	$231_{\times \mathbf{2.1}}$	$142_{\times \mathbf{3.5}}$	365	$83_{\times \mathbf{4.4}}$	$73_{\times \mathbf{5.0}}$
pr2392	3389	–	–	2318	$876_{\times \mathbf{2.6}}$	$528_{\times \mathbf{4.4}}$	2389	$303_{\times \mathbf{7.9}}$	$285_{\times \mathbf{8.4}}$
fnl4461	14817	–	–	11710	–	–	11274	$1073_{\times \mathbf{10.5}}$	$1123_{\times \mathbf{10.0}}$
rl5915	27946	–	–	20935	–	–	20710	–	–

minimizes the total distance. A tour visits each city exactly once (Hamiltonian cycle).

A swap operator for the TSP has been implemented on GPU (2-opt could be implemented in a similar manner). Table 4 shows the results for TSP implementation. On the one hand, even if lots of threads are executed ($\frac{n\times(n-1)}{2}$ neighbors), results for the first configuration are modest (acceleration factor from $\times 1.1$ to $\times 1.5$ for the texture version). Indeed, the incremental evaluation function consists of replacing two to four edges of a solution. As a result, computation of incremental evaluation can be given in constant time, which is not enough to hide the memory latency. Concerning the other configurations, using more multiprocessors overcomes the issue and gives significant results. Indeed, for the GeForce 8800, accelerations starts from $\times 1.7$ with the eil101 instance until $\times 4.4$ for pr2392. In a similar manner, GTX 280 starts from $\times 2.4$ until an acceleration factor of $\times 10$ for the fnl4461 instance.

For larger instances such as pr2392, fnl4461 or rl5915, GPU failed to execute the program because of the hardware register limitation. Another issue concerns the instances of more than 15000 cities (not depicted in the results). For these instances, the number of bytes allocated for the neighborhood fitnesses structure exceeds the memory capacity of the graphics cards. As a consequence, splitting incremental evaluation kernel into several sequential smaller kernels might be considered.

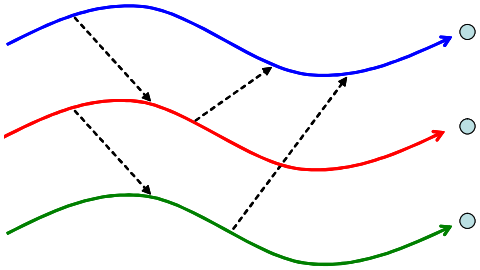

Figure 9: Independent and cooperative walks

6 Multiple Local Search Algorithms on multi-GPU

An extension of the previous approach is to consider the algorithmic-level parallel model for LS algorithms on GPU such as a multistart LS.

6.1 Design on multi-GPU

In the algorithmic-level parallel model, since independent or cooperative self-constained metaheuristics are used (Fig. 9), a multi-GPU approach is well-suited. In the previous design of LS methods on GPU, CPU handled the main process of LS algorithm and GPU operated on the intensive computation part. For multiple LS algorithms, the idea is similar, it consists on associating one LS method per GPU. For a given time, the number of LS algorithms simultaneously executed in parallel is equal to the number of available graphics cards. For achieving this in an efficient way, a mixed multi-core approach must be considered. Indeed, recent multi-core architectures offer an opportunity to make a transition to parallel programming and give more performance in a transparent manner.

As a consequence, since the previous LS design on GPU uses the CPU to manage the LS process and the GPU as a coprocessor, the idea for a multi-GPU approach is to consider one CPU core and one GPU card per LS method. Each CPU process associated to one core executes in parallel the cooperative LS algorithm on GPU previously seen before. Figure 10 illustrates this idea with two cores and two graphics cards. First, the number of process (CPU threads) created is equal to the number of GPU cards. Secondly, each core is associated with a cooperative LS algorithm on GPU (previously designed). Finally, each LS method is executed in parallel, and the results are collected by the CPU for possibly performing operations between the solutions found.

6.2 The Proposed Algorithm on multi-GPU

Algorithm 2 gives a template for multiple LS algorithms on multi-GPU. This template can be seen as an iterative process over an existing LS algorithm on

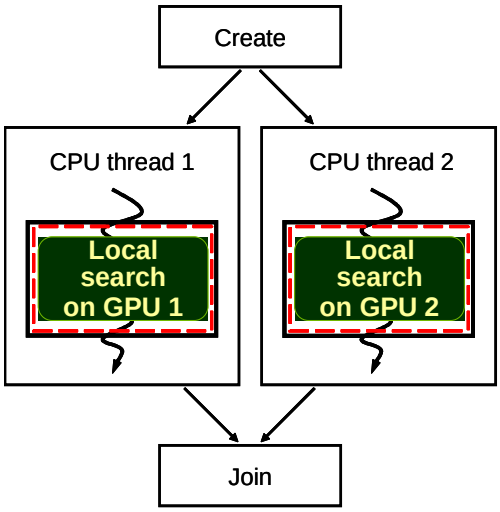

Figure 10: An example of multi-GPU Local Search

GPU. First of all, specific multiple LS treatment has to be made (line 3). Threads

Algorithm 2 Multiple Local Search Template on GPU

```
1:  Specific multiple local search initializations
2:  repeat
3:     Specific multiple local search pre-treatment
4:     Create as many threads as GPU cards
5:     for each thread in parallel do
6:        Execute existing local search template on GPU
7:     end for
8:     Wait all threads to finish
9:     Specific multiple local search post-treatment
10: until a stopping criteria satisfied
```

creation associates one CPU thread (process) to one GPU context (line 4). For doing this, problem specific common structures might be designed for all threads. Secondly, each CPU thread mapped to one graphic card launchs a previously designed single-solution based LS on GPU (lines 5 to 7). Finally, a synchronization barrier must be used for collecting all the results and do some additional post-treatment (lines 8 to 9). The process is repeated until a terminal condition (e.g. a given number of executions).

6.3 Experiments

In order to test the validity of this approach, a multistart local search algorithm has been implemented on the quadratic assignment problem. The well-known

multistart LS is an instantiation of the algorithm-level model, in which different local search algorithms are launched using diverse initial solutions.

The embedded LS algorithm is an iterated tabu search with the same parameters used as before. The configuration used is an Intel Xeon 3Ghz 8 cores with 2 GeForce GTX 280 cards.

6.4 Implementation on multi-GPU

From an implementation point of view, the existing embedded LS algorithm on GPU part remains unchanged. The only thing which remains is to manage CPU cores.

Thread pools (posix and win32 threads) and OpenMP both propose a solution to take full advantage of the use of multi-cores. The basic idea for multi-core programming is to create a set of threads once and for all at the beginning of the program. Indeed, there is a direct mapping between one thread and one CPU core. When a task is created, it executes on a thread in the pool, returning the thread to the pool when the task is done. According to the previous algorithm, the implementation with thread pools or OpenMP is straighforward.

6.5 Results

In the following tables, computational time (in seconds) for different multistart LS algorithms implementations are compared. The sequential version of multistart LS on CPU is used as a reference for the comparaison (acceleration factors). The other implementations are CPU multi-core version, sequential GPU and GPU using texture implementations, and their parallel versions (both multi-core and GPU).

Table 5 gives results for a multistart local search algorithms using 20 subsidiary LS. Since the Intel Xeon is a 8 cores machines, the multiCPU version takes full advantage of all the cores. For smaller instances, overhead creation is more important than computational time. But as long as the size increases, the acceleration factor converges to the expected value $\times 8.0$. Both sequential multistart GPU and GPU texture versions have nearly the same performance than for single-solution based LS algorithm in the previous Tab 1. These implementations become also faster than the CPU version from the instance tai30a (from $\times 1.3$ to $\times 1.8$). Concerning single GPU and multi-GPU versions, the theorical speed-up expected by using two GPU cards instead of one should be $\times 2$. By comparing GPU and MultiGPU columns (respectively GPU_{Tex} and $\text{MultiGPU}_{\text{Tex}}$), from instance tai12a to tai17a, speed-up by using two GPUs is not the one expected due to overhead creation. As the size of the instance grows, speed-up tends to be twice more efficient than a single GPU version. But in practice, the theorical value of acceleration factor is never reached due to thread synchronization, GPU context creation and destruction for each time a process runs a LS algorithm on a GPU card. From the instance tai60a, it becomes efficient to use multi-GPU versions instead of 8 cores stand-alone version (acceleration factors from $\times 11.1$ to $\times 41.3$).

Table 5: Multistart LS Algorithm on QAP (20 embedded LS)

Instance	Intel Xeon 3Ghz 8 cores 20 embedded LS					
	CPU	MultiCPU	GPU	MultiGPU	GPU_{Tex}	$MultiGPU_{Tex}$
tai12a	2.0	$0.4_{\times \mathbf{4.5}}$	$10_{\times 0.2}$	$7.1_{\times 0.3}$	$9.8_{\times 0.2}$	$6.6_{\times 0.3}$
tai15a	3.6	$0.6_{\times \mathbf{6.3}}$	$12_{\times 0.3}$	$8.0_{\times 0.4}$	$11_{\times 0.3}$	$7.1_{\times 0.5}$
tai17a	5.4	$0.7_{\times \mathbf{7.2}}$	$14_{\times 0.4}$	$8.7_{\times 0.6}$	$11_{\times 0.5}$	$7.5_{\times 0.7}$
tai20a	8.8	$1.2_{\times \mathbf{7.5}}$	$16_{\times 0.5}$	$9.7_{\times 0.9}$	$13_{\times 0.7}$	$8.1_{\times \mathbf{1.1}}$
tai25a	17	$2.2_{\times \mathbf{7.7}}$	$19_{\times 0.9}$	$11_{\times \mathbf{1.5}}$	$14_{\times \mathbf{1.1}}$	$9.1_{\times \mathbf{1.9}}$
tai30a	29	$3.8_{\times \mathbf{7.7}}$	$22_{\times \mathbf{1.3}}$	$12_{\times \mathbf{2.3}}$	$16_{\times \mathbf{1.7}}$	$10_{\times \mathbf{2.9}}$
tai35a	46	$6.0_{\times \mathbf{7.9}}$	$24_{\times \mathbf{1.9}}$	$13_{\times \mathbf{3.4}}$	$19_{\times \mathbf{2.5}}$	$11_{\times \mathbf{4.2}}$
tai40a	70	$8.9_{\times \mathbf{7.9}}$	$27_{\times \mathbf{2.6}}$	$15_{\times \mathbf{4.6}}$	$21_{\times \mathbf{3.3}}$	$12_{\times \mathbf{5.8}}$
tai50a	136	$17_{\times \mathbf{8.0}}$	$33_{\times \mathbf{4.1}}$	$18_{\times \mathbf{7.5}}$	$25_{\times \mathbf{5.3}}$	$14_{\times \mathbf{9.4}}$
tai60a	238	$29_{\times \mathbf{8.0}}$	$39_{\times \mathbf{6.0}}$	$21_{\times \mathbf{11.1}}$	$31_{\times \mathbf{7.7}}$	$17_{\times \mathbf{13.9}}$
tai64c	292	$36_{\times \mathbf{8.0}}$	$43_{\times \mathbf{6.8}}$	$23_{\times \mathbf{12.6}}$	$32_{\times \mathbf{8.9}}$	$18_{\times \mathbf{16.1}}$
tai80a	593	$74_{\times \mathbf{8.0}}$	$66_{\times \mathbf{9.0}}$	$35_{\times \mathbf{17.0}}$	$54_{\times \mathbf{10.9}}$	$28_{\times \mathbf{20.5}}$
tai100a	1220	$153_{\times \mathbf{8.0}}$	$110_{\times \mathbf{11.1}}$	$57_{\times \mathbf{21.3}}$	$69_{\times \mathbf{17.5}}$	$36_{\times \mathbf{33.2}}$
tai256c	23738	$2970_{\times \mathbf{8.0}}$	$3169_{\times \mathbf{7.5}}$	$1690_{\times \mathbf{14.0}}$	$1139_{\times \mathbf{20.8}}$	$574_{\times \mathbf{41.3}}$

7 Conclusion and Future Work

A new methodology for LS algorithms on GPU has been made in this paper. To the best of our knowledge, approaches on LS methods have never been proposed before. The idea of this approach is to let the CPU manages the whole LS process and let the GPU be used as a coprocessor for high calculations. This way, the LS parallelism is done at iteration-level: each candidate solution from a given neighborhood is evaluated in parallel in the GPU, and post-treatment can be made on CPU. This model fits well for LS methods such as hill climbing, tabu search, iterated local search, variable neighborhood search ... For large scale instances, intensive computation of evaluation function, and for a large neighborhood set, speed-ups can really become efficient (up to $\times 40$). In a similar manner, a mixed multi-core and multi-GPU approach allows to handle algorithmic-level methods such as a multistart LS algorithm. The fact of combining iteration-level and algorithmic-level of parallelism gives a complete methodology for designing any LS method on GPU.

A next perspective is to use multi-GPU approach in the iteration-level, especially for designing LS methods for tackling larger instances. It will consist of partitioning the neighborhood set, where each partition is executed on a single GPU. That way, multi-GPU approach will allow to increase the speed-up of the exploration space of a given solution. But since each GPU has its own private memory, managing context execution of different GPUs in an efficient way is not a straighforward task.

In the future, all the GPU concepts will be also integrated in the ParadisEO platform. This framework was developped for the design of parallel hybrid meta-

heuristics dedicated to the mono and multiobjective resolution [9]. ParadisEO is a framework based on a clear conceptual separation of metaheuristics concepts, and can be seen as a white-box object-oriented with some reusable concepts. The Parallel Evolving Objects (PEO) module includes well-known parallel and distributed models for metaheuristics such as LS methods. This module will be extended in the future with multi-core and GPU programming techniques.

References

[1] Li, J.M., Wang, X.J., He, R.S., Chi, Z.X.: An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In: Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on. (2007) 855–862

[2] Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: GECCO. (2007) 1566–1573

[3] Wong, T.T., Wong, M.L.: Parallel evolutionary algorithms on consumer-level graphics processing unit. In: Parallel Evolutionary Computations. (2006) 133–155

[4] Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems **22**(2) (2007) 69–78

[5] Talbi, E.G.: From design to implementation. Wiley (2009)

[6] NVIDIA: CUDA Programming Guide Version 2.1

[7] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., mei W. Hwu, W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPOPP. (2008) 73–82

[8] Pointcheval, D.: A new identification scheme based on the perceptrons problem. In: EUROCRYPT. (1995) 319–328

[9] Cahon, S., Melab, N., Talbi, E.G.: Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. J. Heuristics **10**(3) (2004) 357–380

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399